



下载APP



28 | 定向监控：怎样快速发现业务异常？

2021-12-21 高楼

《全链路压测实战30讲》

课程介绍 >

**讲述：高楼**

时长 16:09 大小 14.80M



你好，我是高楼。这节课，我们来讲一讲定向监控。

其实全局监控和定向监控，我在三个专栏里都反复提过。你可能会问了，既然之前都写过了，为什么还要在这个专栏中专门再写一篇呢？毕竟凑字也不是我的风格，我还真是要写点不一样的东西。

在这一节课，我会主要介绍业务级的定向监控，也就是要实现业务代码级的监控。如果你想学习更多定向监控的知识，也可以看看我以前讲过的内容：[🔗 《性能测试场景：如何进行监控设计？》](#) 和 [🔗 《如何设计全局和定向监控策略？》](#)。



话说回来，为什么要做业务级的定向监控呢？因为线上压测是有风险的，这个大家都知道。而做到业务级的定向监控可以降低压测风险。

业务级的定向监控对于刚入行的性能测试小白来说确实是一个新话题，但对于从事多年性能测试老鸟来说并不奇怪。那我们怎么具体落地呢，下面我们就来仔细看一看。

本地环境准备

在之前的课程里，我已经介绍了市场上常见的监控工具，这里我就结合 Prometheus SDK 来说说如何改造代码并落地业务 Metrics 埋点。

Prometheus 提供度量的四种基本类型包括：Counter，Gauge，Histogram，Summary。

这里，我主要演示电商项目下单业务的定向监控。我们会主要采集**订单成功次数、订单失败次数、订单金额总数**等业务的 Metrics，如果你还想扩展其它业务，也可以参考这一方法进行改造。

依照惯例，我还是先在本地搭建 demo 做技术预演，预演成功后再做真实系统改造。

首先，我们要搭建 Prometheus 集成 Grafana 的本地环境。具体的操作你可以参考下面三篇文章：

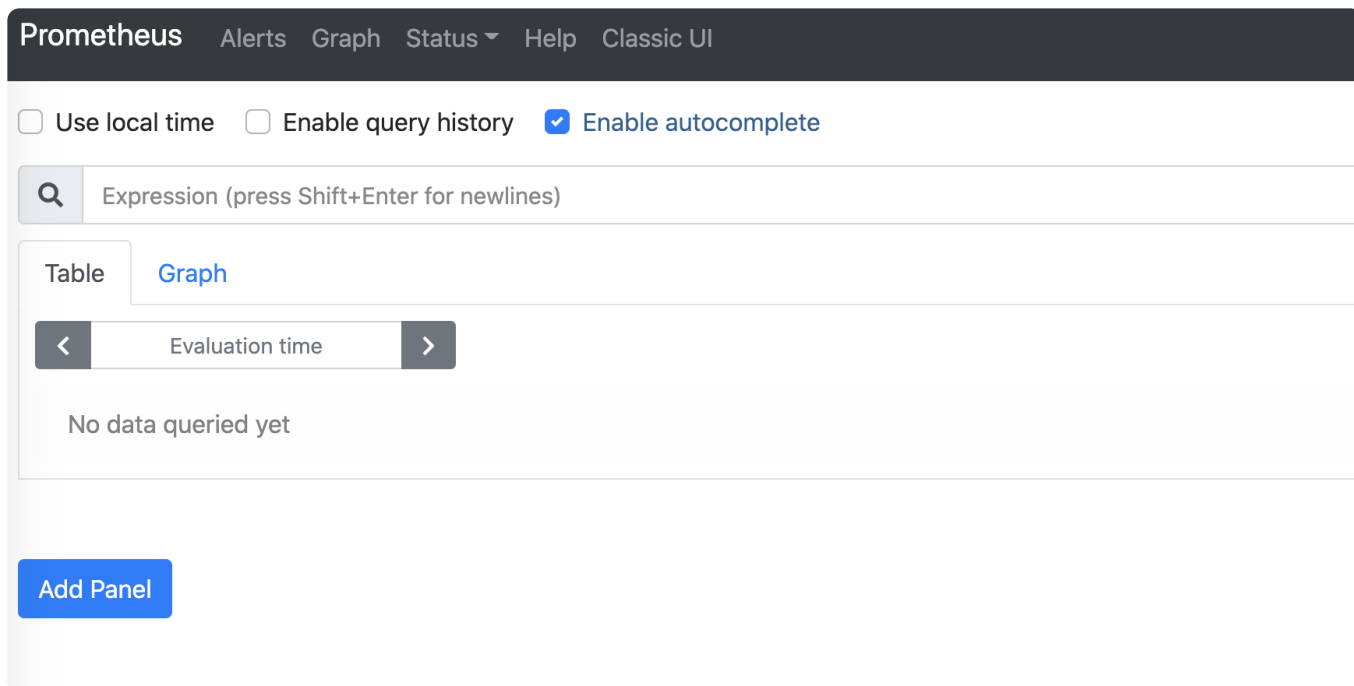
🔗 《性能监控之初识 Prometheus》

🔗 《性能监控之 node_exporter+Prometheus+Grafana 实现主机监控》

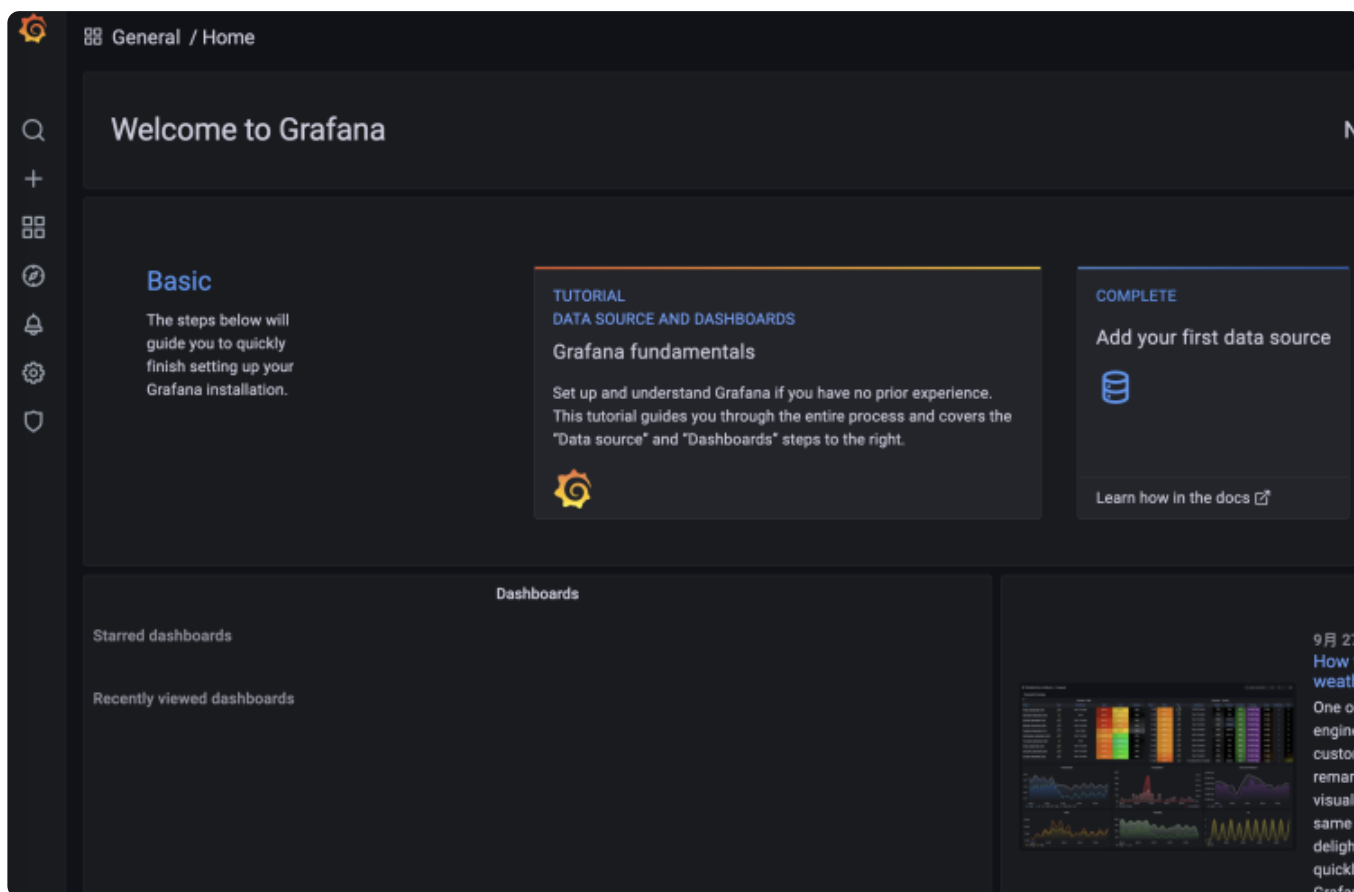
🔗 《性能监控工具之 Grafana+Prometheus+Exporters》

环境搭建好之后，运行效果如下：

Prometheus 监控系统：



Grafana 可视化看板：



这样，我们用 Prometheus 集成的 Grafana 的本地环境就准备好了。

demo 技术预演

下一步，我们就要搭建业务级的定向监控 demo 了，你可以先看下这张逻辑图：



一般情况下，使用 Prometheus SDK 做业务埋点主要有这几个步骤：

1. 创建 Prometheus Metric 数据项，可以创建一个自定义类；
2. 注册定义好的 Metric；
3. 在业务代码中埋点，对 TSDB 数据写入操作；
4. 提供 HTTP API 接口，让 Prometheus 定时来收集 Metric 数据。

这是一个通用的埋点套路，无论是 Golang 还是 Java 应用都是适应的。下面我们来看下它在 demo 项目中的具体实现：

新建 SpringBoot 工程

使用 SpringBoot（项目使用的框架）集成 Prometheus SDK。这里我们要导入相关依赖：

复制代码

```
1 <dependency>
2   <groupId>io.micrometer</groupId>
3   <artifactId>micrometer-registry-prometheus</artifactId>
4   <version>1.7.4</version>
5 </dependency>
```

修改 SpringBoot 全局配置文件

在 application.yml 文件中输入如下配置：

复制代码

```
1 spring:
```

```
2   application:
3     name: dunshan-prometh
4
5   server:
6     port: 8086
7
8   management:
9     endpoints:
10      web:
11        exposure:
12          include: '*'
13      endpoint:
14        health:
15          show-details: always
16      prometheus:
17        enabled: true
18  metrics:
19    tags:
20      application: ${spring.application.name}
```

改好配置文件之后，启动应用服务器，可以看到 Metric 的访问路径和端口号：

```
com.dunshan.prometh.AppPrometheusMain : Starting AppPrometheusMain using Java 1.8.0_291 on ZBMac-C02CQ5R3M.local
com.dunshan.prometh.AppPrometheusMain : No active profile set, falling back to default profiles: default
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8086 (http)
o.apache.catalina.core.StandardService : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.45]
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1097 ms
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.a.e.web.EndpointLinksResolver : Exposing 14 endpoint(s) beneath base path '/actuator'
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8086 (http) with context path ''
com.dunshan.prometh.AppPrometheusMain : Started AppPrometheusMain in 2.356 seconds (JVM running for 3.064)
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
```

我们通过路径和端口号可以得到下面这个地址：

🔗 <http://localhost:8086/actuator/prometheus>。

启动工程，在浏览器中输入这个地址，结果显示如下：

```
# HELP logback_events_total Number of error level events that made it to the logs
# TYPE logback_events_total counter
logback_events_total{application="dunshan-prometh",level="debug",} 0.0
logback_events_total{application="dunshan-prometh",level="error",} 0.0
logback_events_total{application="dunshan-prometh",level="warn",} 0.0
logback_events_total{application="dunshan-prometh",level="trace",} 0.0
logback_events_total{application="dunshan-prometh",level="info",} 7.0
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds{application="dunshan-prometh",} 1.632796844823E9
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the (young) heap memory pool after one GC to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total{application="dunshan-prometh",} 1.41309752E8
# HELP jvm_buffer_count_buffers An estimate of the number of buffers in the pool
# TYPE jvm_buffer_count_buffers gauge
jvm_buffer_count_buffers{application="dunshan-prometh",id="direct",} 1.0
jvm_buffer_count_buffers{application="dunshan-prometh",id="mapped",} 0.0
# HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine process
# TYPE process_cpu_usage gauge
process_cpu_usage{application="dunshan-prometh",} 0.0
# HELP sun_buffer_total_capacity_buffers An estimate of the total capacity of the buffers in the pool
```

可以看到，Actuator Metrics 数据已经正常显示出来了。

配置 Prometheus 拉取数据

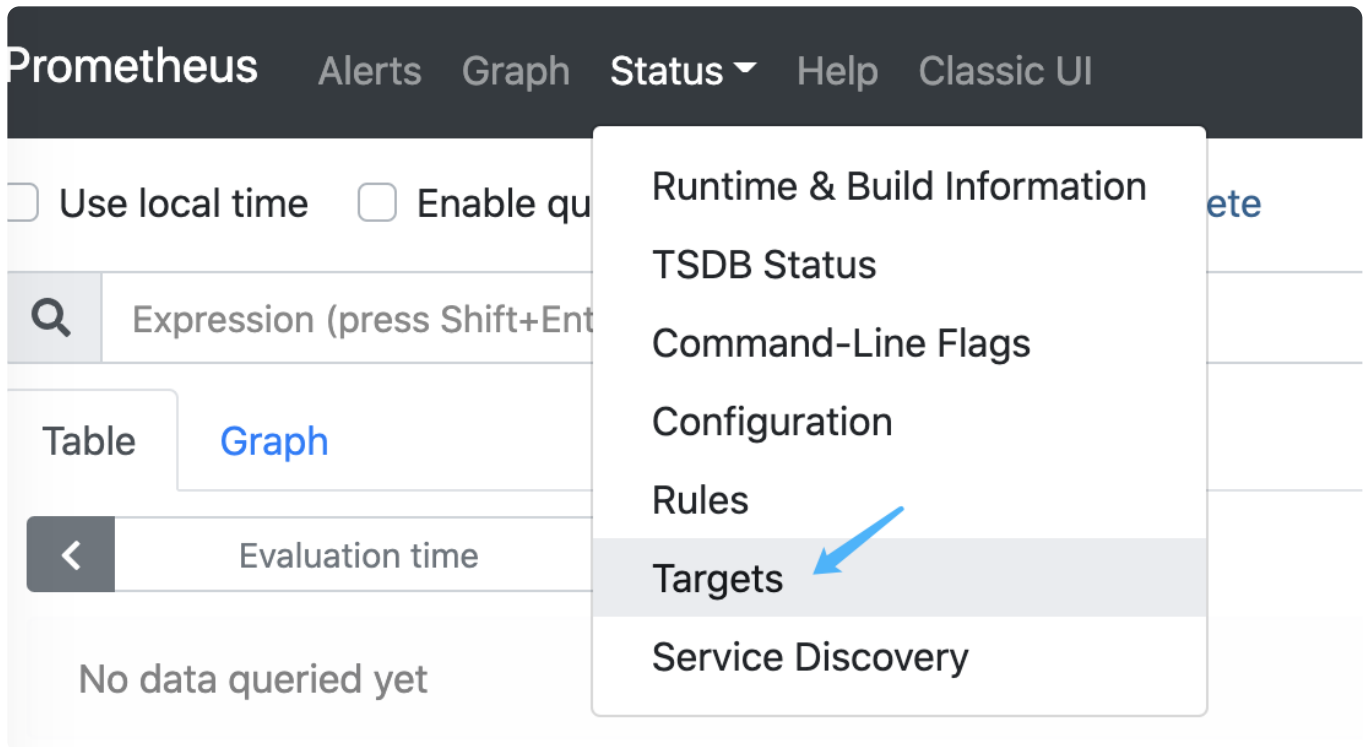
接下来，在 Prometheus 主程序的配置文件中填写上面的 Metrics 接口信息。

在 prometheus.yml 写入的配置你可以参考下面的代码：

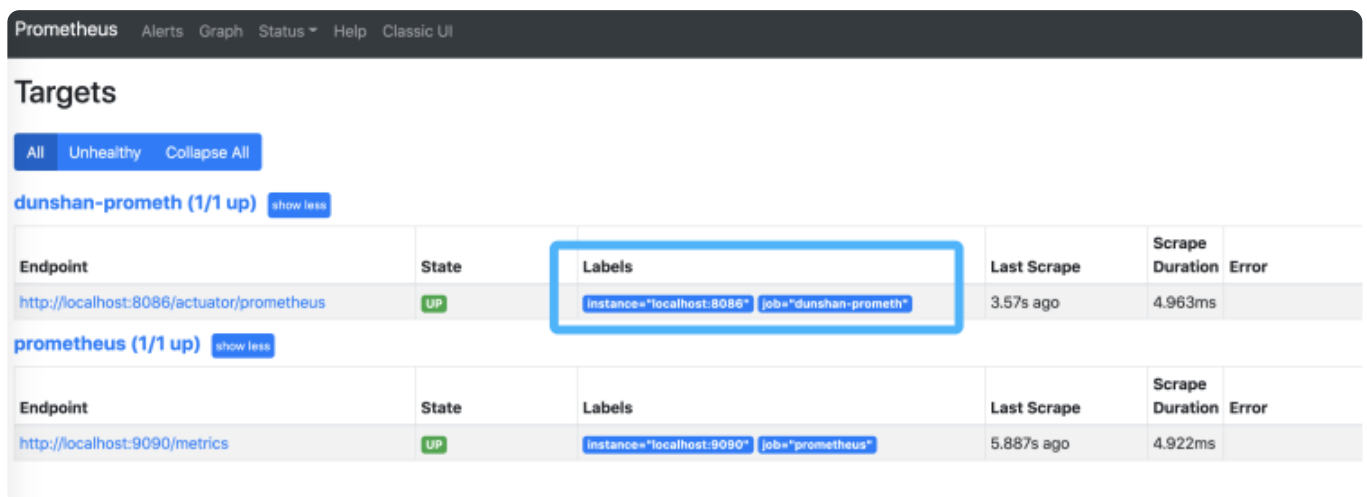
 复制代码

```
1 global:
2   scrape_interval: 15s
3
4   scrape_configs:
5     - job_name: "prometheus"
6       static_configs:
7         - targets: ["localhost:9090"]
8
9     - job_name: "dunshan-prometh"
10       metrics_path: "/actuator/prometheus"
11       static_configs:
12         - targets: ["localhost:8086"]
```

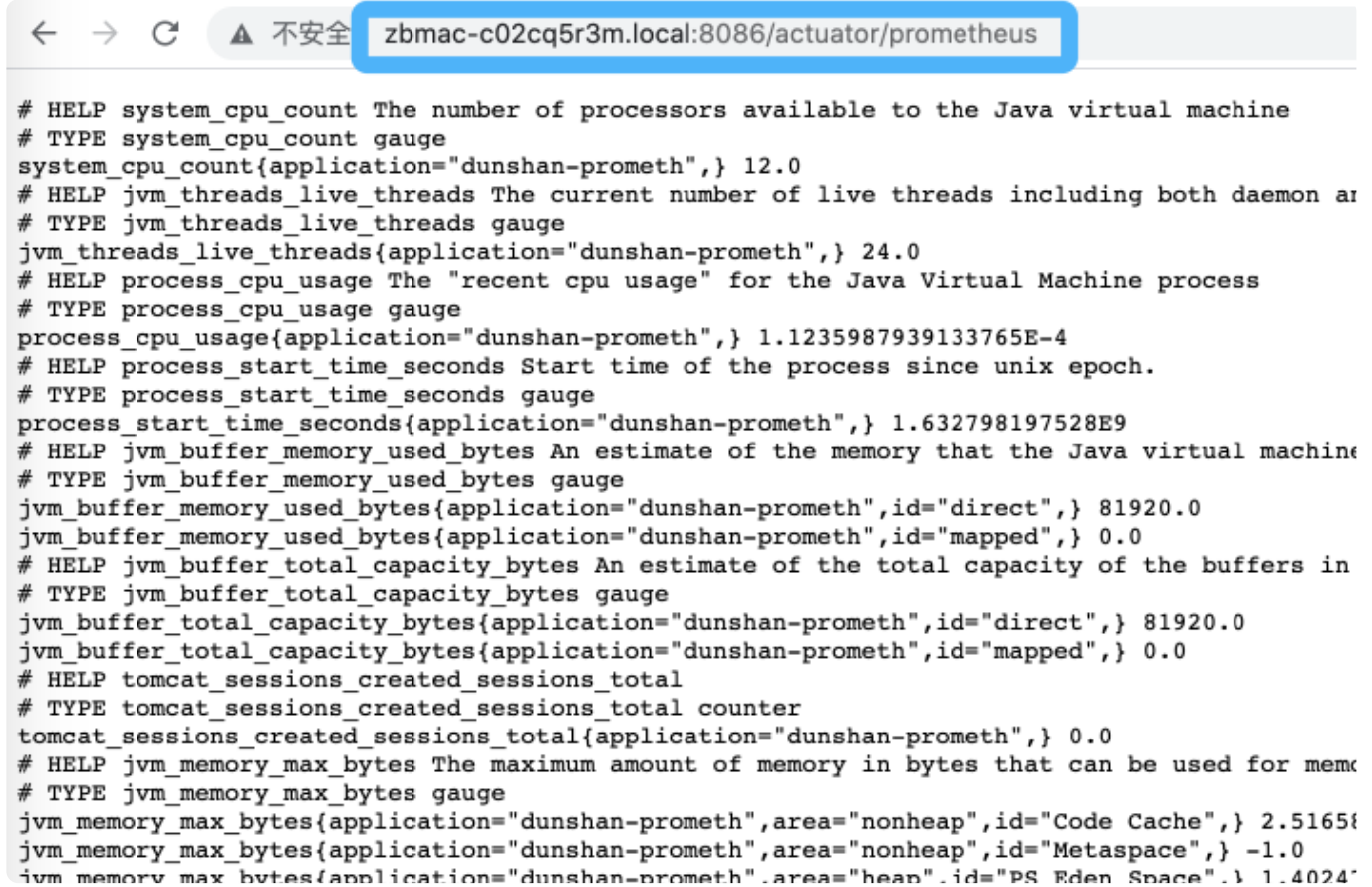
配置完之后，我们重启 Prometheus 服务，在浏览器中输入地址：<http://ip:9090>，在控制台的 Status 菜单栏中选择 Targets：



这样就能看到目前 Prometheus 收集的 Metrics 的路径了：



再次点击 <http://localhost:8086/actuator/prometheus>，显示的信息如下：



The screenshot shows a web browser window with the address bar displaying 'zbmac-c02cq5r3m.local:8086/actuator/prometheus'. The page content lists various Prometheus metrics for the application 'dunshan-prometh'. The metrics include system_cpu_count, jvm_threads_live_threads, process_cpu_usage, process_start_time_seconds, jvm_buffer_memory_used_bytes, jvm_buffer_total_capacity_bytes, tomcat_sessions_created_sessions_total, and jvm_memory_max_bytes. Each metric is accompanied by its type (e.g., gauge, counter) and a numerical value.

```
# HELP system_cpu_count The number of processors available to the Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count{application="dunshan-prometh",} 12.0
# HELP jvm_threads_live_threads The current number of live threads including both daemon and non-daemon threads
# TYPE jvm_threads_live_threads gauge
jvm_threads_live_threads{application="dunshan-prometh",} 24.0
# HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine process
# TYPE process_cpu_usage gauge
process_cpu_usage{application="dunshan-prometh",} 1.1235987939133765E-4
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds{application="dunshan-prometh",} 1.632798197528E9
# HELP jvm_buffer_memory_used_bytes An estimate of the memory that the Java virtual machine has used of the requested buffer memory.
# TYPE jvm_buffer_memory_used_bytes gauge
jvm_buffer_memory_used_bytes{application="dunshan-prometh",id="direct",} 81920.0
jvm_buffer_memory_used_bytes{application="dunshan-prometh",id="mapped",} 0.0
# HELP jvm_buffer_total_capacity_bytes An estimate of the total capacity of the buffers in the Java virtual machine.
# TYPE jvm_buffer_total_capacity_bytes gauge
jvm_buffer_total_capacity_bytes{application="dunshan-prometh",id="direct",} 81920.0
jvm_buffer_total_capacity_bytes{application="dunshan-prometh",id="mapped",} 0.0
# HELP tomcat_sessions_created_sessions_total
# TYPE tomcat_sessions_created_sessions_total counter
tomcat_sessions_created_sessions_total{application="dunshan-prometh",} 0.0
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management.
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{application="dunshan-prometh",area="nonheap",id="Code Cache",} 2.5165E8
jvm_memory_max_bytes{application="dunshan-prometh",area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{application="dunshan-prometh",area="heap",id="PS Eden Space",} 1.4024E9
```

这部分信息告诉我们，应用服务 Metrics 已经被 Prometheus 成功定时收集起来了。

接下来就要在应用服务中埋点收集数据了。在我们的业务服务里面，需要统计的 Metrics 有订单成功次数、订单失败次数和订单金额总数。我们可以通过对订单类中生成的订单号进行埋点来实现。

在项目中新增 PrometheusCustomMonitor 类，初始化业务 Metric。具体的代码参考如下：

 复制代码

```
1 package com.dunshan.prometh.controller;
2
3 import io.micrometer.core.instrument.Counter;
4 import io.micrometer.core.instrument.DistributionSummary;
5 import io.micrometer.core.instrument.MeterRegistry;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Component;
8
9 import javax.annotation.PostConstruct;
10
11 /**
12  * @author dunshan
13  * @program: dunshan-pef
```

```
14  * @description: 自定义下单接口采集类
15  * @date 2021-09-25 10:13:46
16  */
17  @Component
18  public class PrometheusCustomMonitor {
19
20      /**
21       * 下单次数
22       */
23      private Counter orderCount;
24      /**
25       * 下单失败数
26       */
27      private Counter failureCount;
28      /**
29       * 金额统计
30       */
31      private DistributionSummary amountSum;
32
33
34
35      private final MeterRegistry registry;
36
37      @Autowired
38      public PrometheusCustomMonitor(MeterRegistry registry) {
39          this.registry = registry;
40      }
41
42      @PostConstruct
43      private void init() {
44          //统计下单数
45          orderCount = registry.counter("order_request_count", "order", "mall-c
46          //统计失败数
47          failureCount = registry.counter("order_failure_count", "order", "mall
48          //统计金额数据
49          amountSum = registry.summary("order_amount_sum", "orderAmount", "mall
50
51      }
52      public Counter getOrderCount() {
53          return orderCount;
54      }
55
56      public Counter getFailureCount() {
57          return failureCount;
58      }
59
60      public DistributionSummary getAmountSum() {
61          return amountSum;
62      }
63
64  }
```

声明了 Prometheus Metric 类之后，只要在业务方直接调用就可以了。这里我们直接在 Controller 层中增加请求进行模拟测试，从而实现对业务的埋点，收集我们想要的 Metrics 数据。

模拟业务请求

这里我们模拟调用请求，增加订单成功次数、订单失败次数和订单金额总数。

具体代码参考如下：

 复制代码

```
1 package com.dunshan.prometh.controller;
2
3 import com.dunshan.prometh.popj.Result;
4 import lombok.extern.log4j.Log4j2;
5 import org.springframework.stereotype.Controller;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.RequestMapping;
9 import org.springframework.web.bind.annotation.RestController;
10
11 import javax.annotation.Resource;
12 import java.util.HashMap;
13 import java.util.Random;
14
15 /**
16  * @author dunshan
17  * @program: dunshan-pef
18  * @description: 自定义监控采集控制层
19  * @date 2021-09-25 21:33:59
20  */
21 @Log4j2
22 @Controller
23 @RestController
24 public class IndexPromenthController {
25
26
27
28     @Resource
29     private PrometheusCustomMonitor customMonitor;
30
31
32
33     @GetMapping("/")
34     public Result IndexPage() {
35         HashMap<Object, Object> map = new HashMap<>();
```

```
36         map.put("data", "全链路压测业务监控");
37         return Result.buildSuccess(map);
38     }
39
40
41
42     /**
43      * 模拟订单数据
44      *
45      * @param num
46      * @return
47      */
48     @RequestMapping("/order/{num}")
49     public Result orderTotalNumber(@PathVariable int num) {
50         // 统计下单次数
51         if (num == 1) {
52             customMonitor.getOrderCount().increment();
53         } else {
54             //失败数据
55             customMonitor.getFailureCount().increment();
56         }
57         Random random = new Random();
58         int amount = random.nextInt(108);
59         // 统计金额
60         customMonitor.getAmountSum().record(amount);
61         HashMap<String, Object> map = new HashMap<>();
62         map.put("total", amount);
63         map.put("time", System.currentTimeMillis());
64         return Result.buildSuccess(map);
65     }
66
67 }
```

请求完成后，我们在 Prometheus 中查看 Metric 数据，如果数据显示正常，Prometheus TSDB 时序数据库就可以开始定时收集 Metric 数据了。

```
localhost:8086/actuator/prometheus

# HELP tomcat_sessions_created_sessions_total
# TYPE tomcat_sessions_created_sessions_total counter
tomcat_sessions_created_sessions_total{application="dunshan-prometh",} 0.0
# HELP process_files_max_files The maximum file descriptor count
# TYPE process_files_max_files gauge
process_files_max_files{application="dunshan-prometh",} 10240.0
# HELP logback_events_total Number of error level events that made it to the logs
# TYPE logback_events_total counter
logback_events_total{application="dunshan-prometh",level="debug",} 0.0
logback_events_total{application="dunshan-prometh",level="error",} 0.0
logback_events_total{application="dunshan-prometh",level="warn",} 0.0
logback_events_total{application="dunshan-prometh",level="trace",} 0.0
logback_events_total{application="dunshan-prometh",level="info",} 7.0
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{application="dunshan-prometh",area="nonheap",id="Code Cache",} 7492032.0
jvm_memory_used_bytes{application="dunshan-prometh",area="nonheap",id="Metaspace",} 3.4980816E7
jvm_memory_used_bytes{application="dunshan-prometh",area="heap",id="PS Eden Space",} 3.0215488E7
jvm_memory_used_bytes{application="dunshan-prometh",area="heap",id="PS Survivor Space",} 0.0
jvm_memory_used_bytes{application="dunshan-prometh",area="heap",id="PS Old Gen",} 1.3575216E7
jvm_memory_used_bytes{application="dunshan-prometh",area="nonheap",id="Compressed Class Space",} 4886776.0
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the (young) heap memo
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total{application="dunshan-prometh",} 1.1521076E8
# HELP jvm_classes_unloaded_classes_total The total number of classes unloaded since the Java virtual machi
# TYPE jvm_classes_unloaded_classes_total counter
jvm_classes_unloaded_classes_total{application="dunshan-prometh",} 7.0
# HELP jvm_threads_peak_threads The peak live thread count since the Java virtual machine started or peak w
# TYPE jvm_threads_peak_threads gauge
jvm_threads_peak_threads{application="dunshan-prometh",} 28.0
# HELP jvm_threads_daemon_threads The current number of live daemon threads
# TYPE jvm_threads_daemon_threads gauge
jvm_threads_daemon_threads{application="dunshan-prometh",} 2.0
# HELP order_request_count_total
# TYPE order_request_count_total counter
order_request_count_total{application="dunshan-prometh",order="mall-order",} 2.0
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{application="dunshan-prometh",area="nonheap",id="Code Cache",} 2.5165824E8
```

最后，我们只要在 Prometheus 控制台输入表达式语言 PromQL，就能查询 Metric 相关的数据了。

Prometheus Alerts Graph Status Help Classic UI

☐ Use local time ☐ Enable query history ☒ Enable autocomplete

Q

order_request_count_total{application="dunshan-prometh",order="mall-order",}

Table

Graph

<

Evaluation time

>

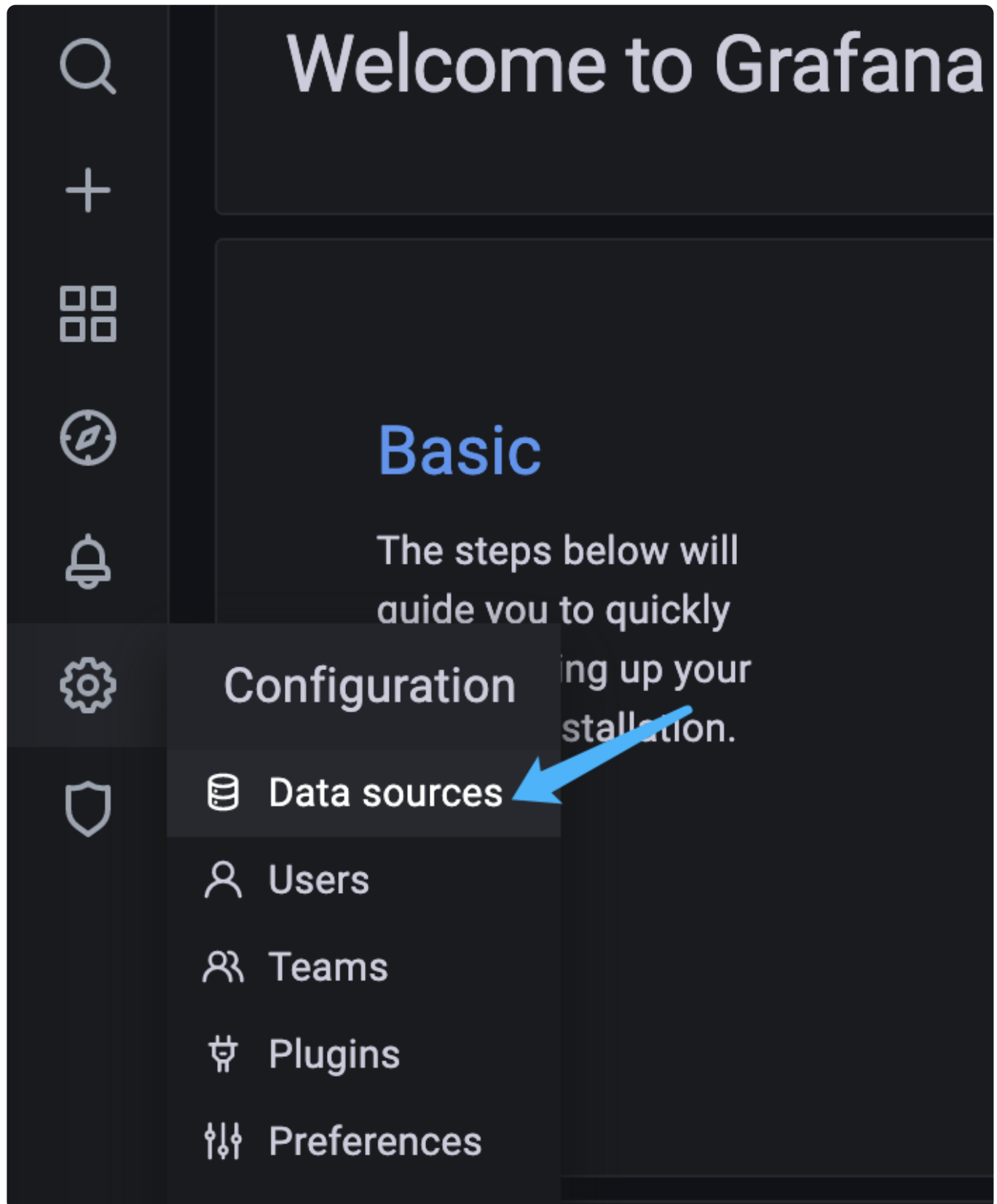
order_request_count_total{application="dunshan-prometh",instance="localhost:8086",job="dunshan-prometh",order="mall-order"}

Add Panel

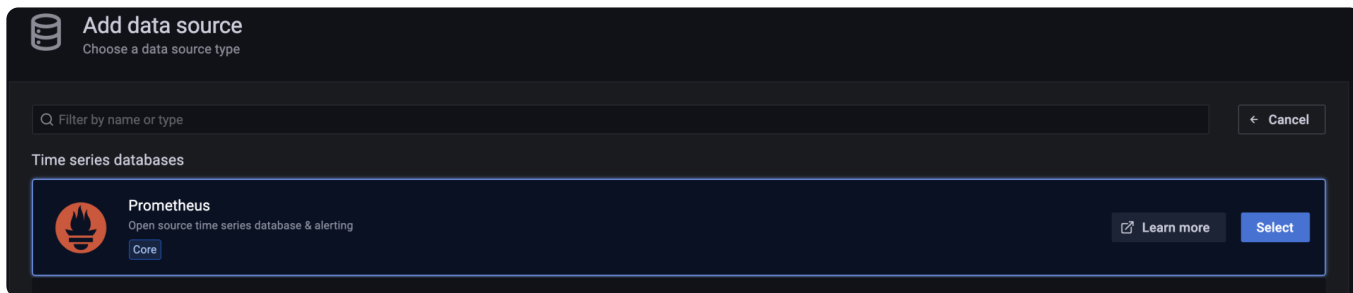
集成 Grafana 数据展示

表达式的结果可以通过多种形式展示，我们可以结合 Grafana 的控件将数据显示为图形或者表格，也可由外部系统通过 HTTP API 调用。这里我主要介绍一下通过集成 Grafana 做可视化数据展示的方法。

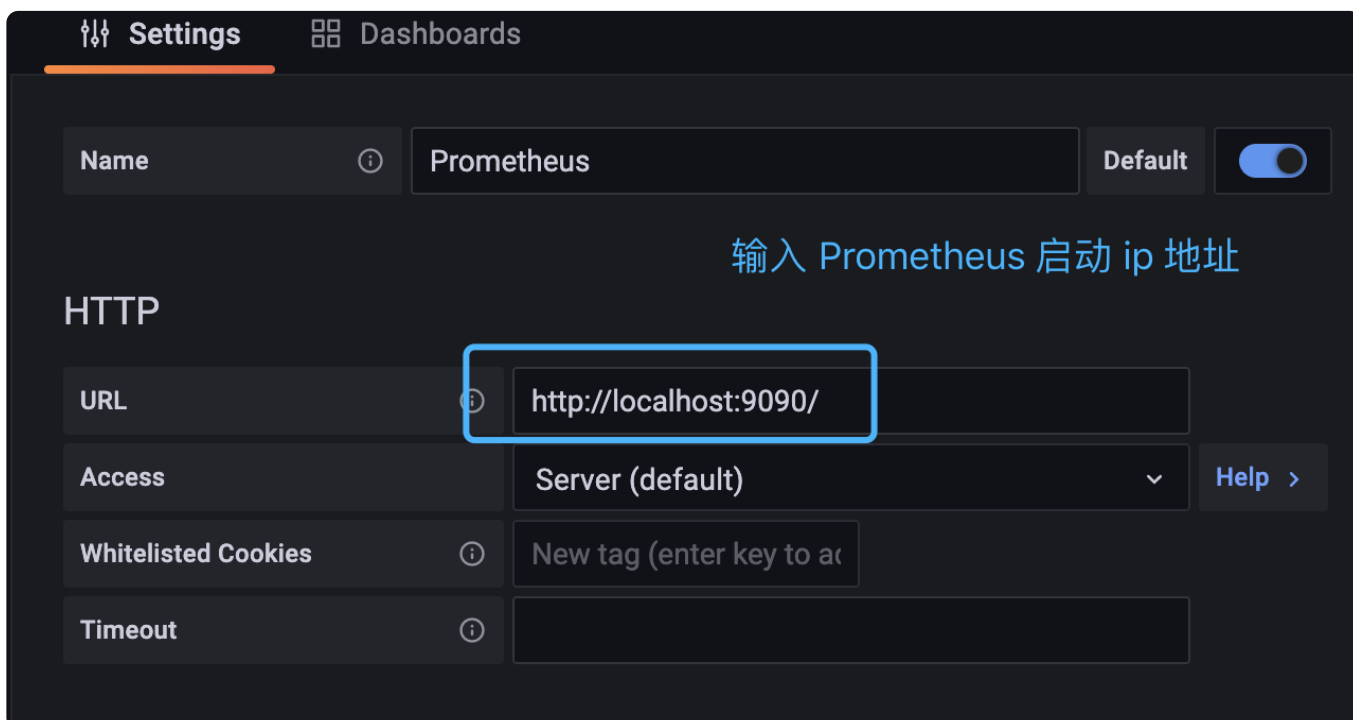
第一步，创建数据源：



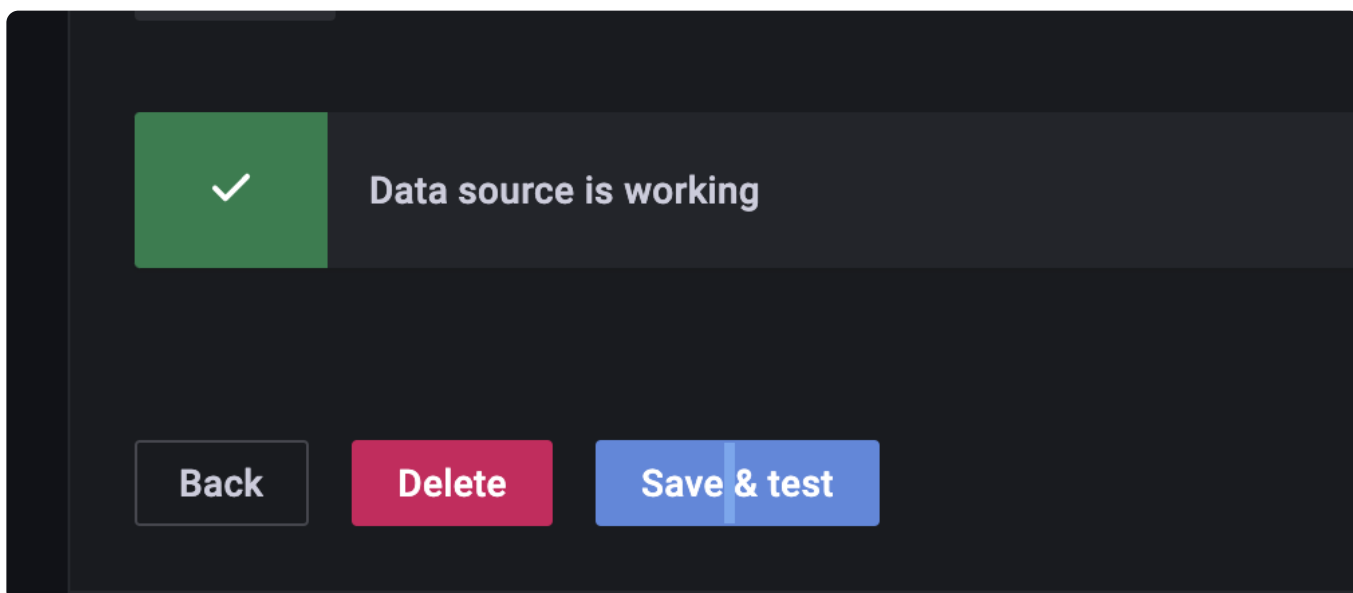
第二步，选择 Prometheus 数据源：



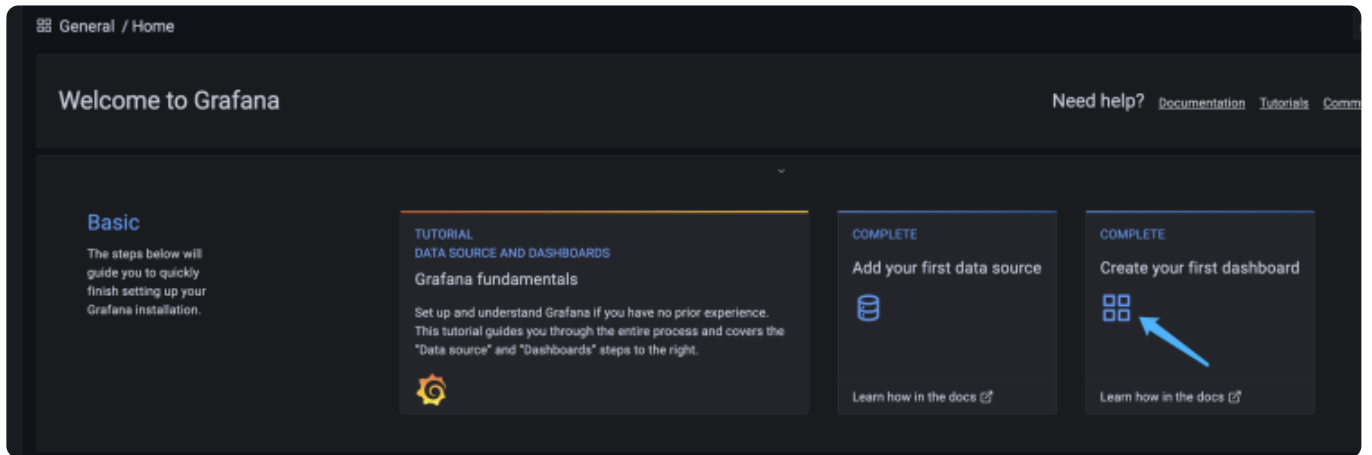
第三步，配置数据源地址：



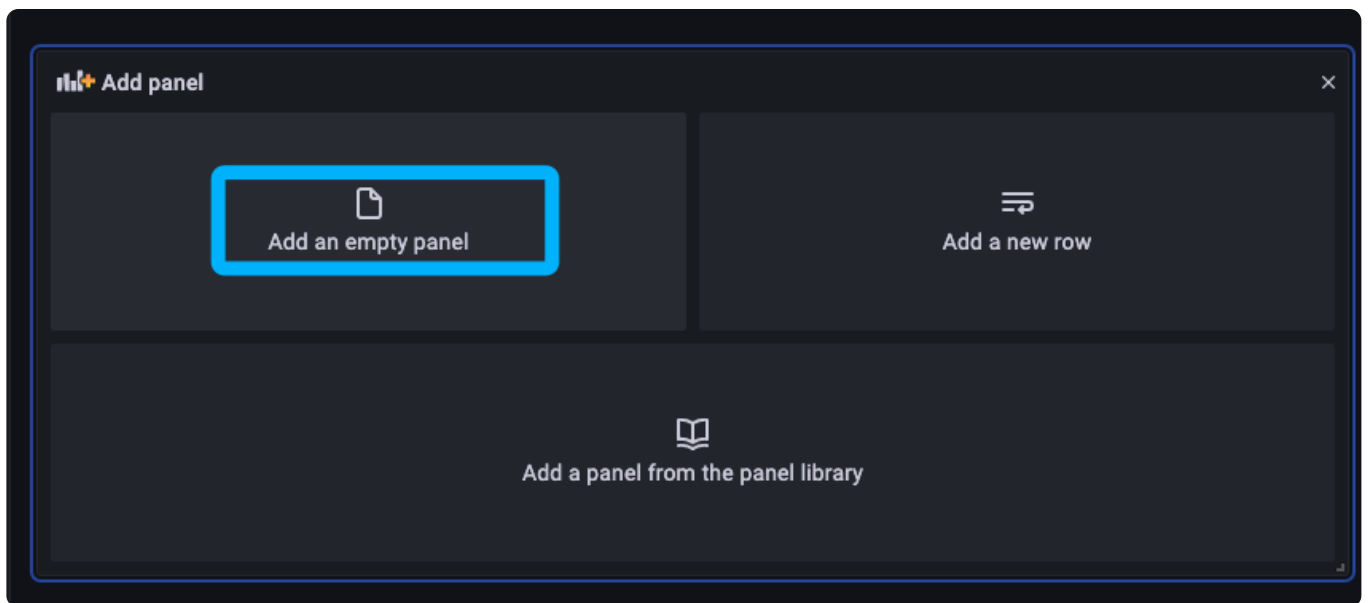
第四步，测试数据源是否配置成功：



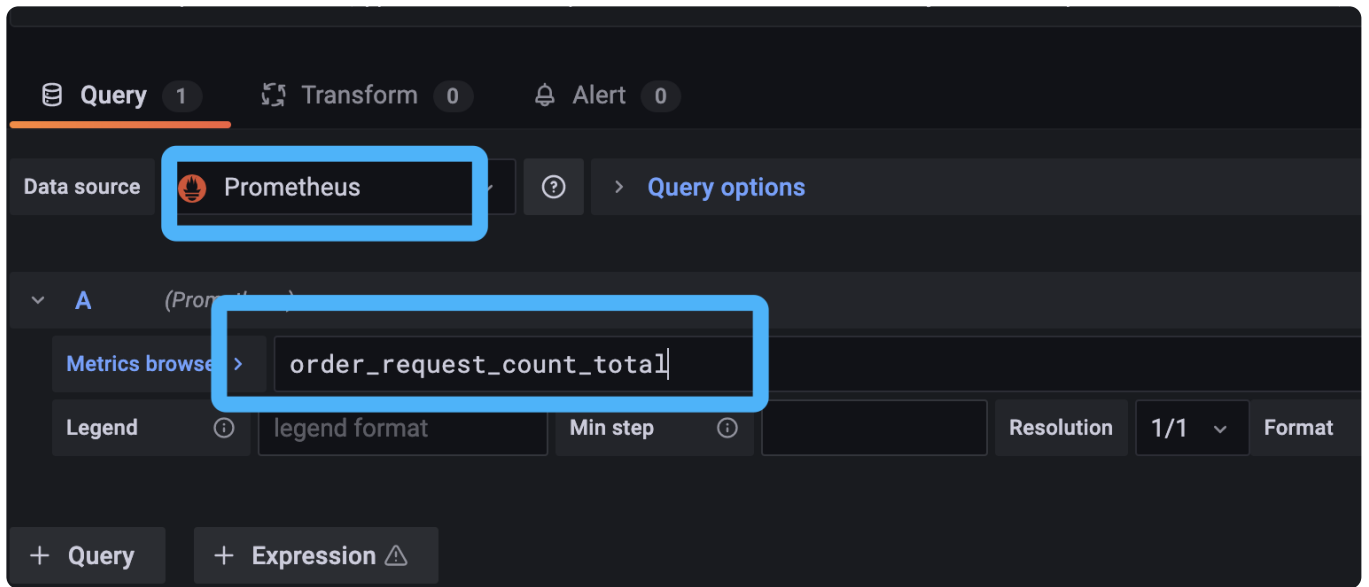
第五步，为了添加 Metrics 的图表，选择“创建第一个 dashboard”：



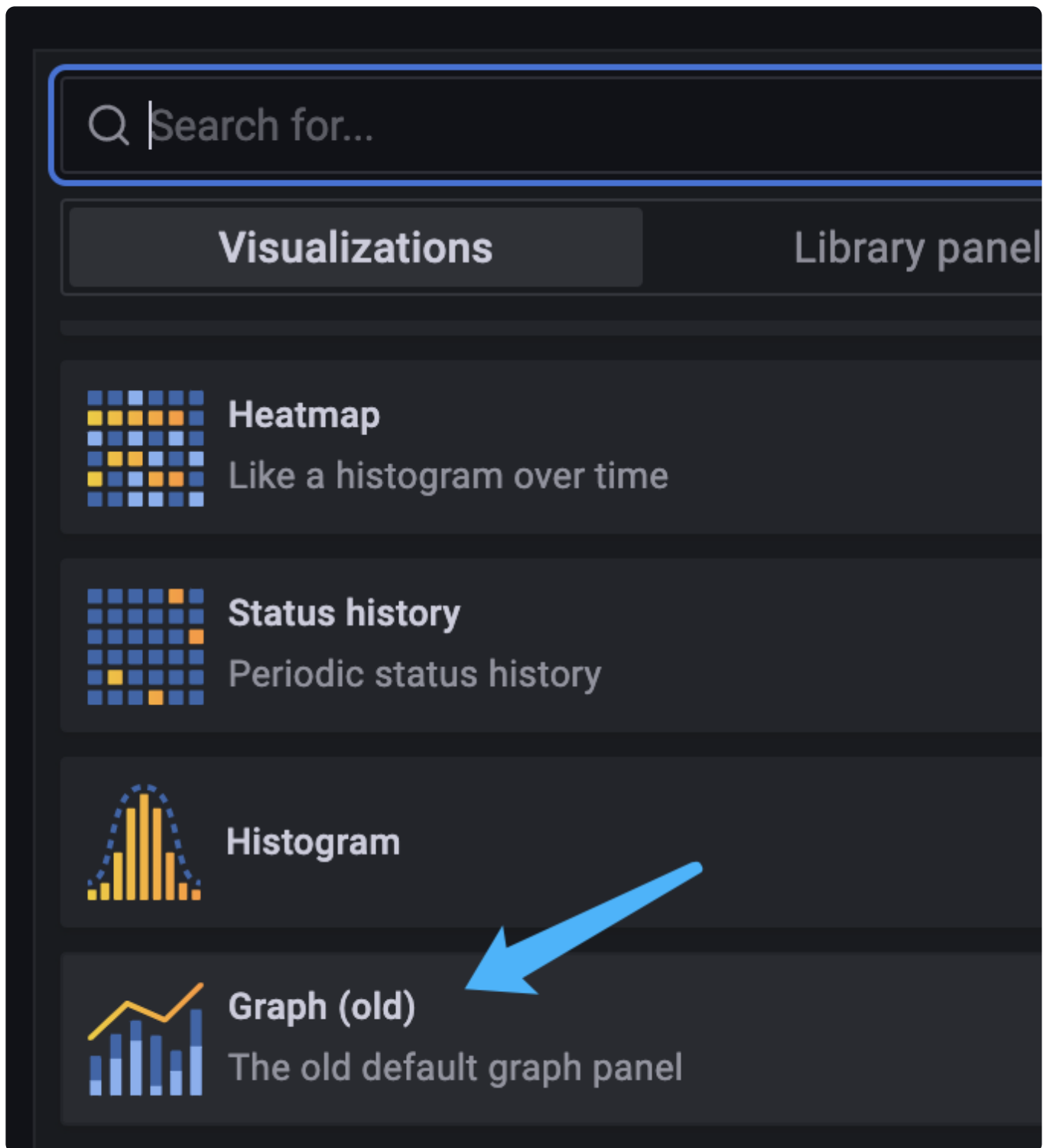
第六步，添加一个空面板：



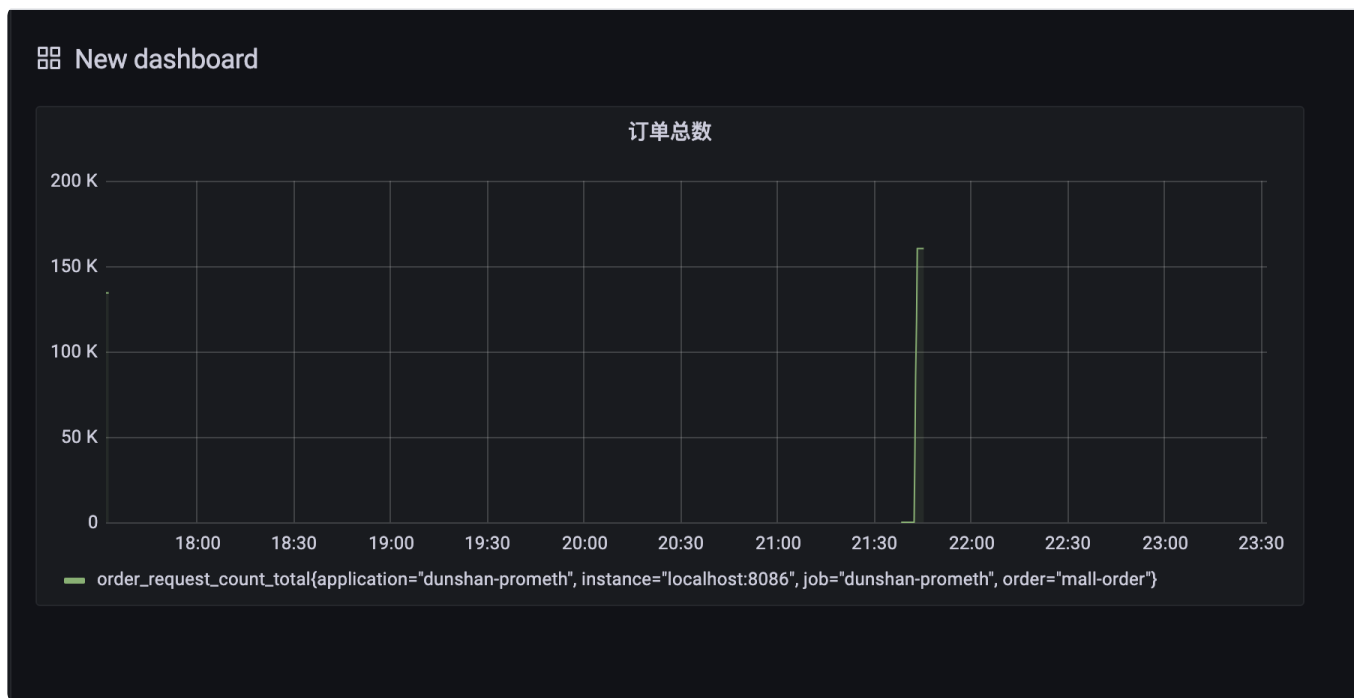
第七步，在 Data source 中选择 Prometheus 数据源，在 Metrics browser 中输入 Metric PromQL 表达式：



第八步，在右侧控件类型中选择图表控件：

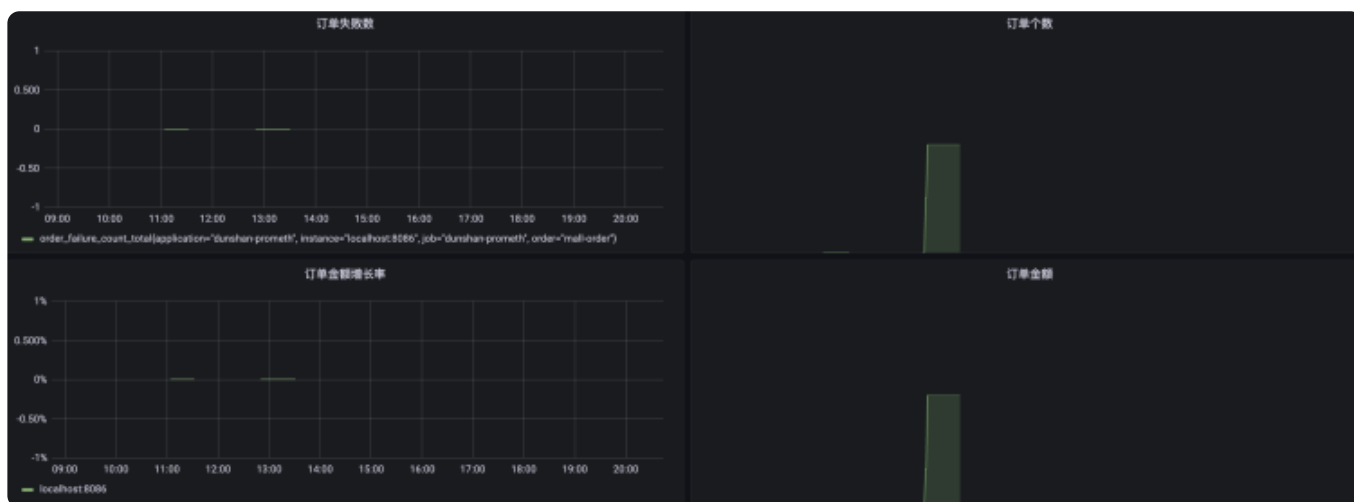


图表创建好之后，我们就可以看到 Metric 数据了。



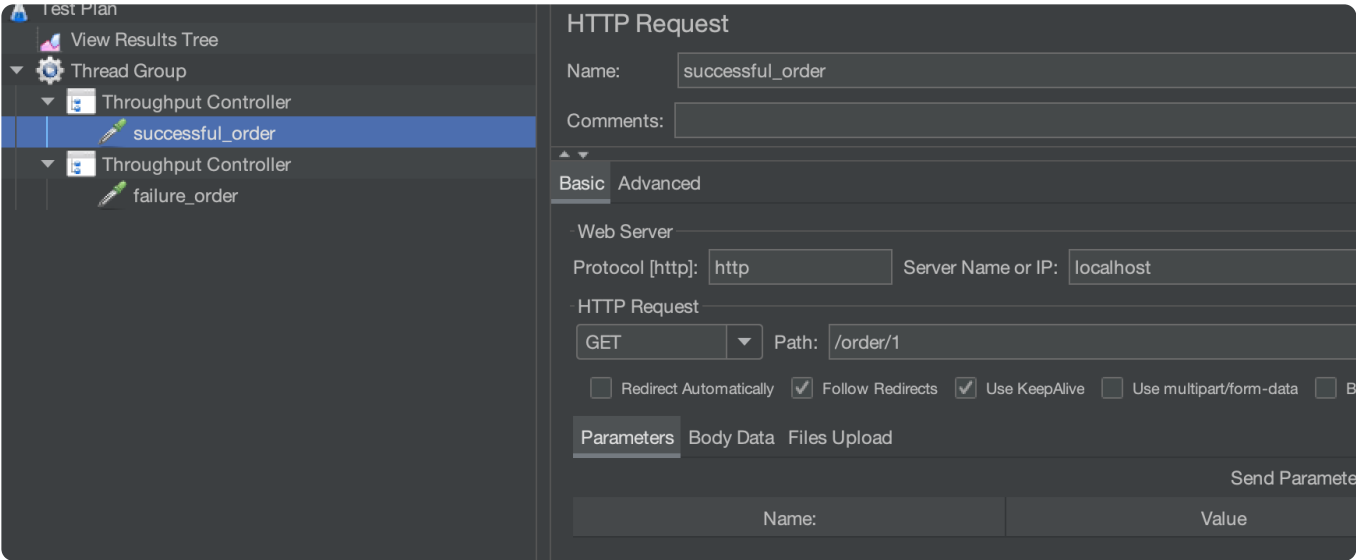
因为需要展示不同的 Metrics，所以我们要重复上面的步骤，增加订单成功次数、订单失败次数、订单金额总数等 Metrics 绑定的图表。

创建好的可视化看板你可以参考下面这张图：



因为在添加可视化看板之前已经有过几次成功的订单，所以我们可以看到，上面已经有数据显示出来了。为了验证订单失败情况下 Metrics 数据是否能正常显示，我们还是使用 JMeter 模拟一下不同的业务情况（区分正确订单与失败订单），这样才能验证业务埋点是否成功。

我们使用 JMeter Throughput Controller 组件分配失败订单与成功订单的比例，脚本配置如下：



执行 JMeter 后，我们通过 Grafana 看板中显示的正常和失败订单的数据，可以知道 demo 的埋点已经成功了：



好了，刚才我们在 demo 技术预演项目中完成了业务模块的埋点工作，接下来就要开始真实系统的改造了。针对订单业务的数据，我们还是要用流量标记区分正常流量和压测流量，这样才能更精准地监控业务情况。当然啦，这也方便我们把这些 Metrics 数据提取出来，后续提供给管理层做决策。

真实系统改造

在对真实系统进行改造之前，我们先梳理一下订单类的代码逻辑，知道在哪里埋点是很重要的。

创建 Prometheus 数据的 Metric

首先还是在项目中新增 Prometheus Metric 类：

 复制代码

```
1  /**
2      * 下单次数
3      */
4      private Counter orderCount;
5      /**
6      * 下单失败数
7      */
8      private Counter failureCount;
9      /**
10     * 金额统计
11     */
12     private DistributionSummary amountSum;
13
14
15
16     /**
17     * 影子表订单数据
18     */
19     private Counter shadowOrderCount;
20
21     /**
22     * 影子表下单失败数
23     */
24     private Counter shadowFailureCount;
25     /**
26     * 影子表金额统计
27     */
28     private DistributionSummary shadowAmountSum;
29
30
31
32     /**
33     * Micrometer 是一个抽象类
34     */
35     private final MeterRegistry registry;
36
37
38
39
40
41     @Autowired
42     public PrometheusCustomMonitor(MeterRegistry registry) {
43         this.registry = registry;
44     }
45
46     @PostConstruct
47     private void init() {
```

```
48     //统计下单数
49     orderCount = registry.counter("order_request_count", "order", "mall-c
50     //统计失败数
51     failureCount = registry.counter("order_failure_count", "order", "mall
52     //统计金额数据
53     amountSum = registry.summary("order_amount_sum", "orderAmount", "mall
54
55     //影子表统计下单数
56     shadowOrderCount = registry.counter("shadow_order_request_count", "or
57     //影子表统计失败数
58     shadowFailureCount = registry.counter("shadow_order_failure_count", "
59     //影子表统计金额数据
60     shadowAmountSum = registry.summary("shadow_order_amount_sum", "orderA
61
62
63
64     }
65     public Counter getOrderCount() {
66         return orderCount;
67     }
68
69     public Counter getFailureCount() {
70         return failureCount;
71     }
72
73     public DistributionSummary getAmountSum() {
74         return amountSum;
75     }
76
77     public Counter getShadowOrderCount() {
78         return shadowOrderCount;
79     }
80
81     public Counter getShadowFailureCount() {
82         return shadowFailureCount;
83     }
84
85     public DistributionSummary getShadowAmountSum() {
86         return shadowAmountSum;
87     }
```

集成 Prometheus SDK

选择订单系统项目，在项目的 pom 文件中添加相关依赖：

 复制代码

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
```

```
3 <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
5 <dependency>
6 <groupId>io.micrometer</groupId>
7 <artifactId>micrometer-registry-prometheus</artifactId>
8 </dependency>
```

注册定义好的业务 Metric

在代码中增加如下配置：

```
@Component
public class PrometheusCustomMonitor {
    /** 下单次数 */
    private Counter orderCount;
    /** 下单失败数 */
    private Counter failureCount;
    /** 金额统计 */
    private DistributionSummary amountSum;
    /** 影子表订单数据 */
    private Counter shadowOrderCount;
    /** 影子表下单失败数 */
    private Counter shadowFailureCount;
    /** 影子表金额统计 */
    private DistributionSummary shadowAmountSum;

    /** Micrometer是一个抽象类 */
    private final MeterRegistry registry;
```

埋点到业务代码

接下来，我们开始梳理 mall-order 服务中生成订单类代码的逻辑。我们可以找到控制器接口代码：

复制代码

```
1 @ApiOperation("根据购物车信息生成订单")
2 @RequestMapping(value = "/generateOrder", method = RequestMethod.POST)
3 @ResponseBody
4 public CommonResult generateOrder(@RequestBody OrderParam orderParam) {
```

```
5     Map<String, Object> result = portalOrderService.generateOrder(orderParam);
6     return CommonResult.success(result, "下单成功");
7 }
```

通过代码调用关系，我们可以很快找到 **GenerateOrder** 方法。业务 **Metric** 需要在这里埋点，以便收集 API 请求数据。

 复制代码

```
1  /**
2   * 根据提交信息生成订单
3   */
4  @Override
5  public Map<String, Object> generateOrder(OrderParam orderParam) {
6      List<OmsOrderItem> orderItemList = new ArrayList<>();
7      //获取用户名
8      UmsMember currentMember = memberService.getCurrentMember();
9      //获取包含促销活动信息的购物车列表
10     List<CartPromotionItem> cartPromotionItemList = cartItemService.listPromot
11
12     .....
13     .....
14     .....
15
16     //发送延迟消息取消订单
17     sendDelayMessageCancelOrder(order.getId());
18     Map<String, Object> result = new HashMap<>();
19     result.put("order", order);
20     result.put("orderItemList", orderItemList);
21     logger.info("结果：" + JSON.toJSONString(result));
22     return result;
23 }
```

因为前面我们针对流量隔离进行系统改造的时候，已经区分了正常流量与压测流量，所以在具体进行业务埋点的时候，我们就需要根据 Header 标记区分不同流量来统计对应的 Metric。

这里我演示一下如何通过 `HttpServletRequest` 来获取 Header 标记来区分不同的流量（也可以使用 AOP 的方式）。参考代码如下：

```
String header = request.getHeader(DataSourceNames.DUNSHAN);
//标识头信息
if (header != null && header.equals(DataSourceNames.HEAD)) {
    if (order.getId() != null) {
        customMonitor.getShadowOrderCount().increment();
    } else {
        customMonitor.getShadowFailureCount().increment();
    }
    customMonitor.getShadowAmountSum().record(order.getTotalAmount().doubleValue());
} else {
    if (order.getId() != null) {
        customMonitor.getOrderCount().increment();
    } else {
        customMonitor.getFailureCount().increment();
    }
    customMonitor.getAmountSum().record(order.getTotalAmount().doubleValue());
}

return result;
```

好了，完成业务代码埋点之后，我们需要提供 Metrics 接口给 Prometheus TSDB 时序数据库收集数据。

公布业务 Metrics 访问路径

修改项目的全局配置文件，公布 Metrics 访问路径。具体配置如下：

```
management: #开启SpringBoot Admin的监控
  endpoints:
    web:
      exposure:
        include: '*'
  endpoint:
    health:
      show-details: always
  prometheus:
    enabled: true
  metrics:
    tags:
      application: ${spring.application.name}
```

然后启动项目观察日志，发现已经有 actuator 对应的 URL 路径了。

```
: Cluster created with settings {hosts=[39.105.21.227:27017], mode=SING
: Opened connection [connectionId{localValue:1, serverValue:1989}] to 3
: Monitor thread successfully connected to server with description Serv
: Exposing 21 endpoint(s) beneath base path '/actuator'
: No URLs will be polled as dynamic configuration sources.
: To enable URLs as dynamic configuration sources, define System proper
: No URLs will be polled as dynamic configuration sources.
: To enable URLs as dynamic configuration sources, define System proper
: Initializing ExecutorService 'applicationTaskExecutor'
: Initializing ExecutorService 'taskScheduler'
: Tomcat started on port(s): 8084 (http) with context path ''
```

在浏览器中访问地址，可以看到，Metrics 数据就都已经显示出来了。

← → ↺ ① localhost:8084/actuator/prometheus

```
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{application="mall-order",area="heap",id="PS Old Gen",} 2.863661056E9
jvm_memory_max_bytes{application="mall-order",area="heap",id="PS Survivor Space",} 2.3068672E7
jvm_memory_max_bytes{application="mall-order",area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{application="mall-order",area="nonheap",id="Code Cache",} 2.5165824E8
jvm_memory_max_bytes{application="mall-order",area="nonheap",id="Compressed Class Space",} 1.073741824E9
jvm_memory_max_bytes{application="mall-order",area="heap",id="PS Eden Space",} 1.376256E9
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{application="mall-order",area="heap",id="PS Old Gen",} 4.9439896E7
jvm_memory_used_bytes{application="mall-order",area="heap",id="PS Survivor Space",} 2.303608E7
jvm_memory_used_bytes{application="mall-order",area="nonheap",id="Metaspace",} 7.6643456E7
jvm_memory_used_bytes{application="mall-order",area="nonheap",id="Code Cache",} 1.8207872E7
jvm_memory_used_bytes{application="mall-order",area="nonheap",id="Compressed Class Space",} 1.021756E7
jvm_memory_used_bytes{application="mall-order",area="heap",id="PS Eden Space",} 3.34548736E8
# HELP zipkin_reporter_messages_bytes_total Total bytes of messages reported
# TYPE zipkin_reporter_messages_bytes_total counter
zipkin_reporter_messages_bytes_total{application="mall-order",} 1754.0
# HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine process
# TYPE process_cpu_usage gauge
process_cpu_usage{application="mall-order",} 0.0036247518307846614
# HELP tomcat_sessions_alive_max_seconds
# TYPE tomcat_sessions_alive_max_seconds gauge
tomcat_sessions_alive_max_seconds{application="mall-order",} 0.0
# HELP tomcat_sessions_active_current_sessions
```

配置 Prometheus 拉取数据

配置 prometheus.yml 配置文件：

📄 复制代码

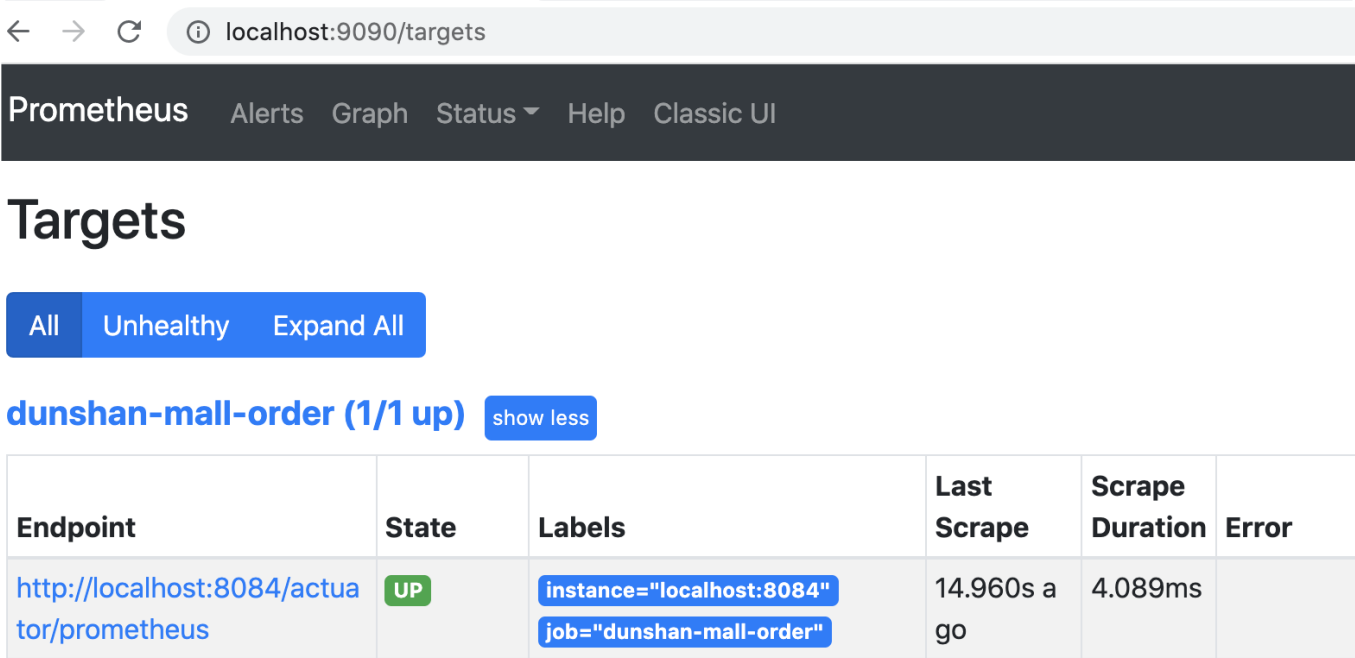
```
1 - job_name: "dunshan-mall-order"
2   metrics_path: "/actuator/prometheus"
3   static_configs:
4     - targets: ["localhost:8084"]
```

文件配置好之后，我们重启 Prometheus，参考命令如下：

📄 复制代码

```
1 $ brew services restart prometheus
2 Stopping `prometheus`... (might take a while)
3 ==> Successfully stopped `prometheus` (label: homebrew.mxcl.prometheus)
4 ==> Successfully started `prometheus` (label: homebrew.mxcl.prometheus)
```

打开 Prometheus 控制台，可以看到，显示的 metrics_path 和我们预期的一致。

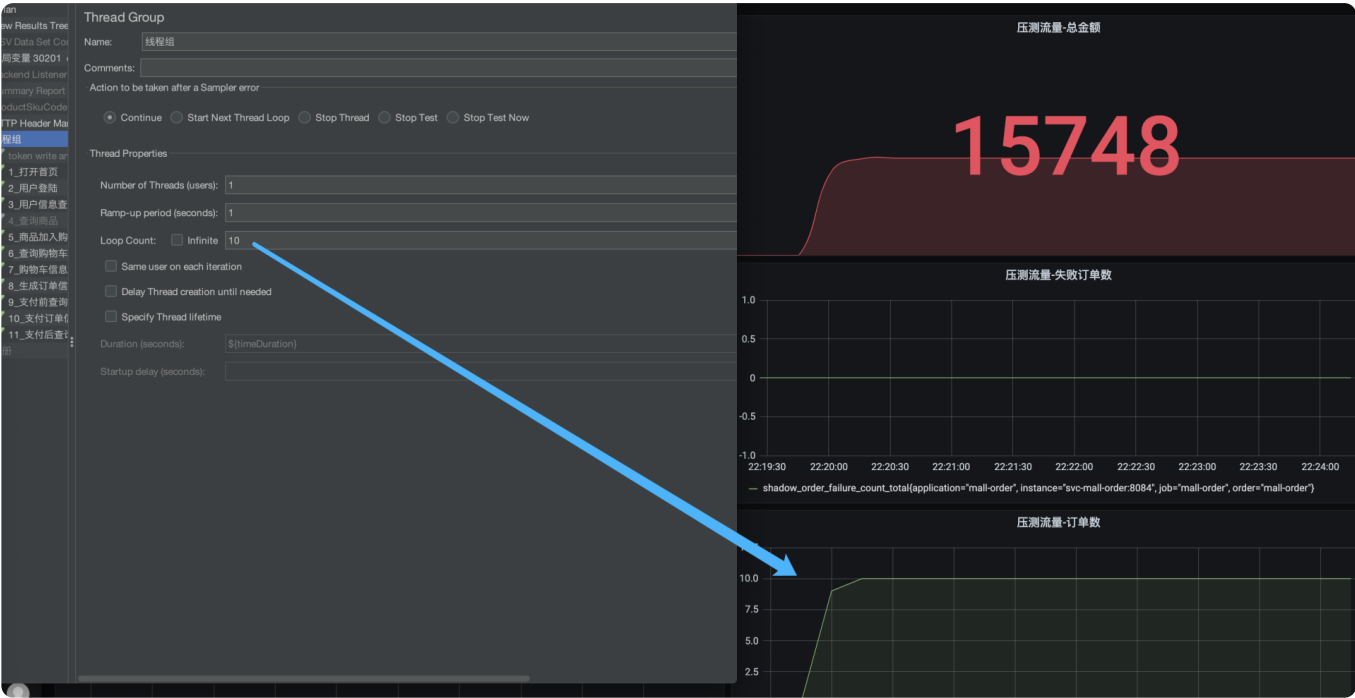
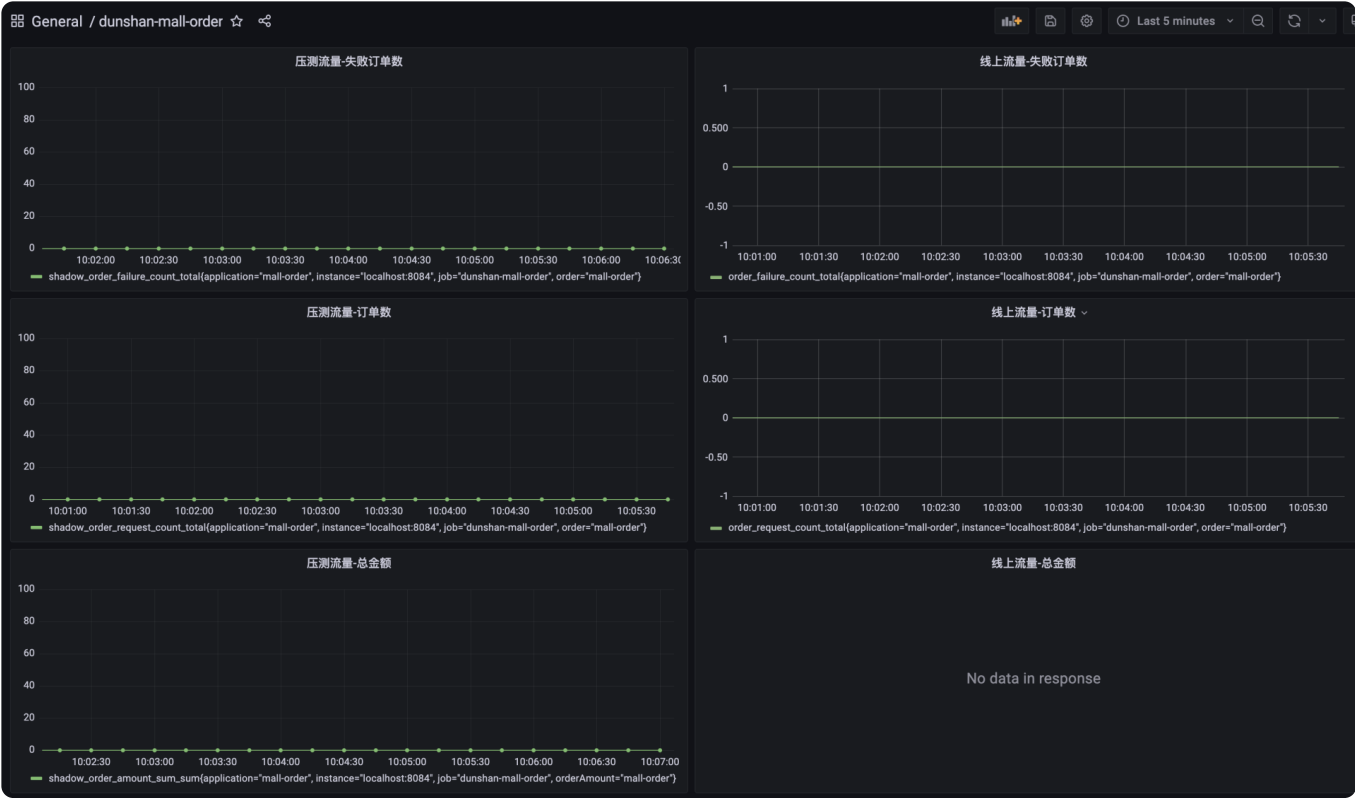


到这里，我们就把订单系统、业务 Metrics 和 Prometheus 监控系统打通了。做完这些我们的任务就完成 80% 了，最后一步就是集成 Grafana，这样才能做到酷炫的可视化数据展示。

集成 Grafana 可视化看板

现在只要打开 Grafana，添加自定义图表绑定 Metrics 即可。

你可以参考下面的运行效果图和最终效果图：



好了，到这里我们的真实系统改造就完成了。

总结

这节课，我们介绍了结合 Prometheus SDK 对 Java 应用的业务埋点改造，我还对这部分细节做了详细的演示。希望可以帮助你在选择业务级定向监控解决方案时有一些参考。

通过这节课的学习，我们知道了使用 Prometheus 可以快速搭建监控服务，它还能够非常方便地在应用程序中进行集成。Prometheus 允许我们通过多种语言的 SDK 进行应用程序的数据埋点。基于这些 SDK，我们可以让应用程序快速纳入到 Prometheus 的监控或者开发者自己的监控数据收集程序当中。同时，这些客户端收集的监控数据，不仅仅支持 Prometheus，还能支持其他的监控工具。从实现的难度上来说，这节课不算太复杂，只要有一点代码功底都可以实现，但这却是非常重要的部分。

我们做性能工作的人经常不擅长用好理解的方式表达业务上的数据，当我们去说 TPS、响应时间、吞吐量、CPU 使用率等测试指标或技术指标时，对非技术行业的业务人员和领导们来说，他们很难理解，也不会有直观的感受。但是如果你说，我们一小时可以处理 2000 万笔下单操作，他们马上就能明白了。这也是我希望借助这节课的内容达到的效果。

做技术工作的时候，你可以非常细致，但是在和不同团队、不同职位的人沟通的时候，我建议你一定要从对方的角度来思考问题，说对方听得懂的话，这样才能让性能工作的价值体现出来。

思考题

学完今天的课程，请你思考下面两个问题：

1. 业务级的定向监控一般会监控哪些 Metrics？你们公司有没有相应的方法来进行监控？
2. 前面，我们还提到了可以用 AOP 埋点收集数据，但是我并没有给出具体的示例。你知道怎么实现吗？

欢迎你在留言区与我交流讨论。当然了，你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

分享给需要的人，Ta 订阅后你可得 **20 元现金** 奖励

 生成海报并分享

 赞 1

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 27 | 全局监控（下）：如何快速落地全局监控？

下一篇 29 | 基准场景：一个案例，带你搞懂基准场景的关键要点

更多学习推荐

Docker 和 K8s 架构核心

6 小时视频课程，带你入门云原生

点击免费领取 



精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。